railinc interview

typescript angular java

http://www.pilottechnologies.net/#/faq

springboot kafka typescript

java:

- spring constant pool. new spring vs quote spring
  - the constant pool is a cache of commonly used objects such as strings, numbers, and other primitive types. The constant pool is used to avoid unnecessary object creation and to improve performance
  - When you use the `new` keyword to create a bean, Spring will create a new instance of the bean every time it's requested.
  - This can be useful when you need a fresh instance of a bean for each request, but it can also be inefficient if the bean is expensive to create.
- interface vs abstract class
  - interfaces and abstract classes are used to define common behaviors and functionalities that can be shared by multiple classes.
  - interface: all has to be abstract method. **implements**
  - abstract: at least one abstract method. **extends**
  - an abstract class is a class that cannot be instantiated and can only be used as a superclass for other classes.
- oop: inheritance abstract poly
  - Encapsulation: Encapsulation is the practice of hiding the internal details of an object and exposing only the necessary information through a public interface. This helps to protect the object's internal state from being modified or accessed by external code, and can improve code maintainability and reuse.
  - Abstraction: Abstraction is the process of representing complex real-world objects or systems in simplified, abstract terms. It allows us to focus on the essential features of an object or system and ignore the irrelevant details. Abstraction is often achieved through the use of abstract classes, interfaces, and inheritance.
  - Inheritance: Inheritance is the process by which one class (the subclass) inherits properties and behaviors from another class (the superclass). This allows subclasses to reuse code from the superclass and add new functionality as needed. In Java, inheritance is achieved using the `extends` keyword.
  - Polymorphism: Polymorphism is the ability of an object to take on multiple forms or behaviors. In Java, polymorphism is achieved through method overloading (defining multiple methods with the same name but different parameters) and method overriding (defining a method in a subclass that has the same signature as a method in the superclass).

- solid principle
    - Single Responsibility Principle (SRP): This principle states that a class should have only one reason to change. In other words, a class should be responsible for only one thing. This helps to make code easier to understand, test, and maintain, since each class has a clear and focused responsibility.
    - Open/Closed Principle (OCP): This principle states that software entities (classes, modules, etc.) should be open for extension but closed for modification. In other words, we should be able to add new functionality to a system without modifying existing code. This can be achieved through techniques such as abstraction and inheritance.
    - Liskov Substitution Principle (LSP): This principle states that subtypes should be substitutable for their base types. In other words, if a function takes an object of a certain type, it should be able to work with any subtype of that type without any unexpected behavior. This requires careful design of inheritance hierarchies and adherence to the contracts defined by base types.
    - Interface Segregation Principle (ISP): This principle states that clients should not be forced to depend on interfaces they don't use. In other words, we should create small, focused interfaces that define only the methods that are needed by clients. This helps to avoid unnecessary coupling and simplifies code maintenance.
    - Dependency Inversion Principle (DIP): This principle states that high-level modules should not depend on low-level modules, but both should depend on abstractions. In other words, we should depend on abstractions (interfaces, abstract classes, etc.) rather than concrete implementations. This helps to make code more flexible, reusable, and testable.
- override overload
    - Method overriding is the process of defining a method in a subclass that has the same name, return type, and parameters as a method in its superclass
    - Method overloading is the process of defining multiple methods with the same name but different parameters.
- exception's structure. checked unchecked's example
    - extend a exception class
    - checked: must be handled by a catch, or declared in the throw clause. file not found exception. IO exception
    - unchecked: arithmetic excception
- java 8 new feautres: map, stream, optional, function interface, default ma, lamda, groupBy, faltmap, parallel stream,
    - Lambda expressions: Lambda expressions are a shorthand notation for defining anonymous functions in Java. They allow us to write more concise and readable code by eliminating boilerplate code. Lambda expressions are defined using the `->` operator and can be used with functional interfaces.
    - Stream API: The Stream API is a new feature in Java 8 that allows us to process collections of data in a declarative and functional way. It provides a set of operations that can be chained together to transform, filter, and aggregate data. Streams are designed to work efficiently with large data sets and can be processed in parallel.
    - Optional: Optional is a new class in Java 8 that provides a way to represent null values in a more expressive and safe way. It is a container object that may or may not contain a non-null value.

Optional provides methods for checking whether a value is present and for retrieving the value if it is.
    - Default methods: Default methods are a new feature in Java 8 that allow us to add methods to an interface without breaking existing implementations. Default methods provide a default implementation for a method in an interface, which can be overridden by implementing classes.
    - Function interface: The Function interface is a new interface in Java 8 that represents a function that takes one argument and returns a result. It is a generic interface that can be used with any data type, and it provides a way to define functions using lambda expressions.
    - Map: The Map interface is an updated version of the existing Map interface in Java. It provides new methods for working with maps, such as `forEach()`, `replace()`, and `compute()`. The new methods make it easier to work with maps in a functional and declarative way.
    - GroupBy: The groupBy operation in the Stream API allows us to group elements in a collection based on a specified key. It returns a Map object that maps each distinct key to a list of elements that have that key.
    - FlatMap: The flatMap operation in the Stream API allows us to flatten a stream of streams into a single stream. It is useful when working with nested collections and allows us to avoid using nested loops.
    - Parallel stream: The Stream API supports parallel processing of data using the `parallelStream()` method. Parallel streams can split the data into smaller chunks and process them in parallel, which can improve performance on multi-core systems.
- interface has default method, why use abstract, since abstract can maintain status
    - abstract classes can have state, while interfaces cannot.
    - When we need to share code between related classes: Abstract classes can contain common implementation code that can be shared by multiple subclasses. This can help to reduce code duplication and simplify the implementation of related classes.
    - When we need to define non-static methods: Abstract classes can define non-static methods that can be called by subclasses. This can be useful when we need to define common behavior that is not tied to a specific instance of the class.
    - When we need to maintain state: Abstract classes can have instance variables that can be used to maintain state across method calls. This can be useful when we need to track information about the object's state over time.
- thread: thread pool, feuter object to handle return
    - A thread pool is a group of pre-initialized threads that are available to perform a set of tasks. A thread pool can help to reduce the overhead of creating and destroying threads for each task by reusing threads that have already been created. This can improve the performance of applications that perform a large number of short-lived tasks.
    - In Java, the `Executor` framework provides a way to create and manage thread pools. The `Executor` interface defines a single method, `execute()`, which takes a `Runnable` object as an argument and executes it using a thread from the pool. The `ExecutorService` interface extends `Executor` and provides additional methods for managing the thread pool, such as `submit()`, which returns a `Future` object representing the result of the task.
    - A `Future` object is a placeholder for the result of a computation that has not yet been completed. It provides a way to retrieve the result of a computation that is running in a separate thread.

In Java, the `Future` interface defines methods for checking whether the computation is complete, retrieving the result of the computation, and canceling the computation. The `Future` interface is used extensively in conjunction with the `ExecutorService` interface to manage asynchronous tasks.

spring boot:

- 3-4 advantages"
  - tomcat
  - auto configuration
  - Quick and Easy Setup: Spring Boot provides a quick and easy way to create Spring-based applications with minimal configuration. It includes a range of pre-built templates, plugins, and starters that enable developers to get started quickly and focus on writing application logic rather than boilerplate code.
  - Lightweight and Modular: Spring Boot is designed to be lightweight and modular, which means that you can use only the features you need for your application. This helps to reduce the overall size of the application and improve performance.
  - Auto Configuration: Spring Boot includes a powerful auto-configuration feature that automatically configures the Spring framework and third-party libraries based on the dependencies of your application. This saves developers from the hassle of manually configuring these dependencies, which can be time-consuming and error-prone.
  - Built-in Monitoring and Management: Spring Boot includes built-in monitoring and management tools that enable developers to monitor and manage their applications from a centralized dashboard. This makes it easy to monitor the performance and health of the application, as well as to identify and troubleshoot issues in real-time.
  - it is easy to configure and deploy Spring Boot applications to Tomcat.
- Rest APi 要会写
  - consume
  - get/put/put/pos
  - CRUD a product
  - 写到service，写到repository， public interface extends jpa repository。
  - @entity @ column.
  - Consuming data: In a REST API, clients can send data to the server using HTTP requests. The data can be sent in different formats, such as JSON or XML. To consume data in a REST API, you can use the `@RequestBody` annotation in Spring to convert the data to a Java object.
  - HTTP methods: RESTful APIs use HTTP methods such as GET, POST, PUT, and DELETE to interact with resources. The GET method is used to retrieve data, POST is used to create new resources, PUT is used to update existing resources, and DELETE is used to delete resources.
  - CRUD operations: CRUD stands for Create, Read, Update, and Delete. In a RESTful API, these operations can be used to interact with resources. For example, you can use the POST method to create a new resource, the GET method to retrieve a resource, the PUT method to update a resource, and the DELETE method to delete a resource.
  - Service and Repository layers: In a Spring-based REST API, it is common to use a Service layer to implement business logic and a Repository layer to interact with a database or other data source. The

Service layer can call methods on the Repository layer to retrieve or update data, and can perform additional logic such as data validation or formatting.
    - Entity and Column annotations: In a Spring-based REST API, you can use the `@Entity` annotation to define a JPA entity that represents a database table, and the `@Column` annotation to map entity properties to table columns. These annotations provide a way to define the structure and relationships of data in the database.
    - **复习一个spring boot project**
- error code : 2xx 3xx 4xx 5xx
    - 1xx: Informational
        - 100 (Continue)
    - 2xx: Success
        - 200 (OK)
    - 3xx: Redirection
        - 302 (Found) is used to indicate that the requested resource has moved temporarily to a different location, and the client needs to request the resource from that new location.
    - 4xx: Client errors
        - 400 (Bad Request)
        - 404 (Not Found)
    - 5xx: Server errors
        - error on the server while processing the request. The most common code in this class is 500 (Internal Server Error),
        - A 504 error The **upstream** server is offline or not responding.
- security 往知道的上套
    - spring security rewrite the filter(web security)
    - basic authentication: encryption. sh-256, bcrypt
    - jwt token
        - timestamp will be expired
        -
    - Oauth
        - do not use third-party on financial app
        - any third-party has potential leakage
    - keypass
    - Spring Security: Spring Security is a framework that provides authentication, authorization, and other security features for web applications. It is built on top of the Spring framework and provides a range of security features such as user authentication, authorization, session management, and more. Spring Security can be customized and extended to fit the specific security requirements of an application.

    - Basic Authentication: Basic authentication is a simple authentication scheme that involves sending a username and password to the server in plain text. To improve the security of basic authentication, the password can be encrypted using a hash function such as SHA-256 or bcrypt.
    - JWT Token: JSON Web Tokens (JWT) are a standard for representing claims securely between two parties. JWTs can be used to provide authentication and authorization for web applications. JWTs are usually signed using a secret key or public/private key pair, and include a timestamp to indicate when the token will expire.
    - OAuth: OAuth is a protocol for authentication and authorization that is commonly used to allow users to authenticate with third-party services without sharing their credentials with the service. OAuth is often used to allow users to log in to web applications using their social media or other third-party accounts. It is important to note that using third-party authentication services can introduce potential security risks, so it is important to carefully evaluate the risks and benefits of using OAuth in a particular application.
    - KeePass: KeePass is a free and open-source password manager that can be used to store and manage passwords securely. KeePass uses strong encryption algorithms to protect passwords and other sensitive data, and provides features such as password generation and auto-type to improve security and convenience.
- dependency injection: allows objects to be constructed with their dependencies injected at runtime rather than being tightly coupled to their dependencies.
    - Prototype: In the context of Spring, a bean can be defined as a singleton or a prototype. Singleton beans are created once and reused throughout the application, while prototype beans are created each time they are requested. Prototype beans are useful when you need to create a new instance of a bean each time it is used.
    - @Autowired: The @Autowired annotation is used to inject dependencies into a bean. When a bean is created, Spring automatically searches the application context for a bean that matches the type of the dependency and injects it into the bean.
    - How to do DI: There are several ways to perform DI in a Spring application. The most common way is to use the @Autowired annotation to inject dependencies into a bean. Another way is to use XML configuration to define dependencies and wire them together at runtime. Yet another way is to use Java configuration, where the configuration is defined in code rather than XML.
- AOP
    - Aspect-oriented programming (AOP) is a programming paradigm that enables the separation of concerns in a software system. In Spring, AOP is used to separate cross-cutting concerns such as logging, security, and transactions from the core business logic of an application. AOP is implemented using advice, which is a piece of code that is executed at specific points in the application's lifecycle.
- How to handle exception
    - Exception Handling: Exception handling is an important aspect of any application, and Spring provides several mechanisms for handling exceptions in a structured way. One common way to handle exceptions in Spring is to use the @ControllerAdvice annotation to define a global exception handler that applies to all controllers in an application. Another way is to use the @ExceptionHandler annotation to define an exception handler method for a specific controller.

- transaction: Transaction management is an important aspect of any application that involves database operations. In Spring, transactions are managed using the @Transactional annotation. When a method is annotated with @Transactional, Spring creates a transaction for that method and all methods called from within that method. If any exception is thrown during the execution of the method, the transaction is rolled back and all changes made to the database are undone.
    - add @transactional, any failure in any step, all operations will be rolled back
    - propogation: the flow
        - required(default): use transaction in the transaction pool
        - requirednew: have to create
    - isolation:
        - read uncommited
        - read commited
    - roll back for
    - @Transactional: The @Transactional annotation is used to mark a method as transactional. When a method is annotated with @Transactional, Spring creates a transaction for that method and all methods called from within that method. If any exception is thrown during the execution of the method, the transaction is rolled back and all changes made to the database are undone.
    - Propagation: Propagation refers to the behavior of a transaction when it calls another method that is also transactional. There are several propagation options available in Spring, including:
    - REQUIRED: Use the existing transaction if there is one, otherwise create a new transaction.
    - REQUIRES_NEW: Always create a new transaction, even if there is an existing transaction.
    1. Isolation: Isolation refers to the degree to which the changes made by one transaction are visible to other transactions. There are several isolation levels available in Spring, including:
    - READ_UNCOMMITTED: Changes made by one transaction are visible to other transactions before they are committed.
    - READ_COMMITTED: Changes made by one transaction are visible to other transactions only after they have been committed.
    - Rollback: Rollback refers to the behavior of a transaction when an exception is thrown. By default, Spring rolls back the transaction when a runtime exception is thrown. You can customize the rollback behavior using the @Transactional annotation's rollbackFor and noRollbackFor attributes.

```
@Transactional(propagation = Propagation.REQUIRED, isolation = Isolation.READ_COMMITTED)
public void transferFunds(Account fromAccount, Account toAccount, double amount) {
    fromAccount.withdraw(amount);
    toAccount.deposit(amount);
}
```

unit test

SOAP is a protocol used for exchanging structured data between applications over the internet. SOAP stands for Simple Object Access Protocol and it uses XML as its data format. Here are some key concepts related to SOAP:

1. XML: XML is a markup language used for encoding data in a format that is both human-readable and machine-readable. In SOAP, XML is used to define the structure and format of the data being exchanged between applications.
2. WSDL: The Web Services Description Language (WSDL) is an XML-based language used for describing the structure of a web service. A WSDL document defines the methods that can be called on a web service, as well as the input and output parameters for each method.
3. Port: In SOAP, a port is a combination of an endpoint and a binding. An endpoint is a URL that specifies the location of a web service, while a binding specifies the protocol and data format used to communicate with the web service.

database/SQL:

- employee and department: find each department's total salary and display >= 5000.

```
SELECT departments.name AS department, SUM(employees.salary) AS total_salary
FROM employees
INNER JOIN departments ON employees.dept_id = departments.id
GROUP BY departments.name
HAVING SUM(employees.salary) >= 5000;
```

- groupby, sum, count, orderBy
- find the second highest salary
    - limit, offset

```
SELECT DISTINCT salary FROM employee ORDER BY salary DESC OFFSET 1 LIMIT 1
```

```
SELECT MAX(salary) FROM employee WHERE salary < (SELECT MAX(salary) FROM employee);
```

356: **SQL** Find employee whose salary is higher than its own manager (only given employee table which contains employee id and its manager id)

```
SELECT e.name AS employee_name, e.salary AS employee_salary, m.name AS manager_name,
m.salary AS manager_salary
FROM employees e
INNER JOIN employees m ON e.manager_id = m.id
WHERE e.salary > m.salary;
```

- self join: http://www.plottechnologies.net/#/faq 356

kafka:

concept

- Producer: A Kafka producer is a client application that sends messages to a Kafka broker. Producers can send messages to one or more topics, and each message consists of a key and a value.
- Consumer: A Kafka consumer is a client application that reads messages from a Kafka broker. Consumers can read messages from one or more topics, and each message is read from a particular partition.
- Kafka Broker: A Kafka broker is a server in a Kafka cluster that manages one or more partitions of one or more topics. Each broker stores a portion of the topic's data, and messages are replicated across multiple brokers to provide fault tolerance and high availability.
- Topic: A topic is a category or feed name to which messages are published. Topics are partitioned for scalability and can be replicated across multiple brokers for fault tolerance.
- Partition: A partition is a unit of parallelism in Kafka, and each topic can have one or more partitions. Partitions enable horizontal scaling and allow multiple consumers to read from a topic concurrently.
- Consumer Group: A consumer group is a set of one or more consumers that work together to consume messages from a topic. Each consumer in a group reads from a separate partition, and a partition can only be read by one consumer in a group at a time.
- Partition ID: Each partition in a Kafka topic is assigned a unique integer identifier called the partition ID. Partition IDs are used by producers to send messages to specific partitions, and by consumers to read messages from specific partitions.
- Suppose you have a distributed application that generates log messages from multiple servers. You want to collect these log messages in a central location and process them in real-time. To do this, you could use Kafka to create a topic called `logs`, with multiple partitions to enable horizontal scaling. Each server in the application would be a Kafka producer that sends log messages to the `logs` topic, and a group of Kafka consumers would read the log messages from the topic, process them in real-time, and store them in a database. The consumers would be organized into a consumer group to ensure that each message is read by only one consumer, and Kafka's fault tolerance and high availability features would ensure that log messages are not lost even if one or more Kafka brokers go down.
- Ordering: To ensure that data is ordered, you can use partitions. Each partition in a Kafka topic is ordered, so if you have a single partition for a topic, the order of messages is guaranteed. If you have multiple partitions, messages may be out of order across partitions, but within a partition, the order is guaranteed.
- Loss prevention: To prevent data loss, you can use acknowledgement (ACKs) from the consumer to the producer. When a consumer receives a message, it sends an acknowledgement back to the producer to confirm that the message has been received. If the producer does not receive an acknowledgement within a specified time period, it can retry sending the message. In Kafka, you can configure the level of acknowledgement to use when sending messages:
    - acks=0: The producer does not wait for any acknowledgement. This means that messages may be lost if a broker or network failure occurs.

if a broker or network failure occurs.
    - acks=1: The producer waits for acknowledgement from the leader replica (the first replica that receives the message). This provides some level of durability, but messages may still be lost if the leader replica fails before replicating the message to all replicas.
    - acks=all: The producer waits for acknowledgement from all in-sync replicas (ISRs) before considering the message sent. This provides the highest level of durability, but may have higher latency since all replicas must acknowledge the message.
- Duplication prevention: To prevent duplicate messages, you can use message deduplication. One common approach is to add a unique identifier to each message, such as a UUID or a sequence number. Consumers can then use this identifier to detect and discard duplicate messages. In Kafka, you can also configure the consumer to only read new messages from the topic, skipping over any messages it has already processed.

security:

- SASL Authentication: Kafka supports Simple Authentication and Security Layer (SASL) authentication, which enables authentication of clients using various authentication mechanisms, such as Kerberos or username/password. SASL can be used to authenticate both producers and consumers of Kafka messages.
- Access Control Lists (ACLs): Kafka also supports Access Control Lists (ACLs), which are used to control access to Kafka resources. ACLs define which users or groups have access to which topics or partitions. ACLs can be used to restrict access to certain topics or partitions, or to prevent unauthorized modifications to Kafka configurations.
- Encryption: Kafka supports encryption of data in transit and at rest. Data in transit can be encrypted using SSL/TLS, while data at rest can be encrypted using disk encryption or file-level encryption. Encryption ensures that data is protected from unauthorized access, and prevents sensitive data from being intercepted or viewed by unauthorized users.
- Decryption: When encrypted data is consumed from Kafka, it needs to be decrypted before it can be used. This can be done by the consumer, or by an external decryption service. If decryption is done by the consumer, it needs to have the necessary decryption keys or credentials to decrypt the data.

typescript:

how to call backend:

- use a httpclient. get/post/put
- `HttpClient` module from the `@angular/common/http` package to make HTTP requests to a backend API. Here's an example of how to use the `HttpClient` to make GET, POST, and PUT requests:

```
import { HttpClient } from '@angular/common/http';

// Inject the HttpClient service in your component or service
constructor(private http: HttpClient) {}

// Make a GET request
```

```
// Make a GET request
this.http.get('/api/endpoint').subscribe((response) => {
  console.log(response);
}, (error) => {
  console.error(error);
});

// Make a POST request with data
const data = { name: 'John', age: 30 };
this.http.post('/api/endpoint', data).subscribe((response) => {
  console.log(response);
}, (error) => {
  console.error(error);
});

// Make a PUT request with data
const data = { name: 'John', age: 30 };
this.http.put('/api/endpoint', data).subscribe((response) => {
  console.log(response);
}, (error) => {
  console.error(error);
});
```

In these examples, we first inject the `HttpClient` service in our component or service using dependency injection. We then use the `get()`, `post()`, and `put()` methods of the `HttpClient` to make GET, POST, and PUT requests, respectively. These methods take the URL of the API endpoint as their first argument, and an optional payload of data as their second argument.

For each request, we use the `subscribe()` method to handle the response and any errors. The `subscribe()` method takes two callback functions: one for handling the successful response, and another for handling any errors.

Note that when making POST or PUT requests, you may need to set additional headers or options, such as the content type. You can do this using the `HttpHeaders` class or the `HttpParams` class from the `@angular/common/http` package.

how to receive aysnchronous

- Using callbacks: You can use callbacks to handle the response from the backend API. This involves passing a callback function as an argument to the HTTP request method, which will be called with the response data when it is available.

```
this.http.get('/api/endpoint', (response) => {
  console.log(response);
});
```

- Using Promises: Promises provide a way to handle asynchronous operations in a more structured way. You can use the `toPromise()` method of the `HttpClient` to convert the HTTP response into a Promise. For example:

```
this.http.get('/api/endpoint').toPromise().then((response) => {
  console.log(response);
}).catch((error) => {
  console.error(error);
});
```

- Using Observables: Observables are similar to Promises, but they provide more flexibility and are often used in reactive programming. You can use the `subscribe()` method of the `HttpClient` to subscribe to an Observable and handle the response data. For example:

```
this.http.get('/api/endpoint').subscribe((response) => {
  console.log(response);
}, (error) => {
  console.error(error);
});
```

angular

- how to display object in html. use {} two way binding
- only when something is true, display somehthing: NGIF
- NGfor

security

- authguard
- canActivate

route/routing

BQ

- Sprints: You organize your development work into time-boxed iterations, usually lasting two weeks, called sprints.
- Planning Meeting: At the start of each sprint, you have a planning meeting where you meet with the business stakeholders to identify the features and requirements that will be developed during the sprint.
- Standup Meetings: During the sprint, you have daily standup meetings where the development team discusses progress, roadblocks, and priorities.
- Sprint Review and Retrospective: At the end of each sprint, you have a review meeting to demonstrate the completed work to the stakeholders and get feedback, followed by a retrospective meeting to discuss what went well and what can be improved for the next sprint

went well and what can be improved for the next sprint.

In addition to these key practices, Agile development also emphasizes collaboration, communication, and flexibility. If conflicts arise between group members or leaders, it's important to communicate openly and find a resolution that best serves the project's goals. Code review is another important practice that can help ensure quality and consistency in the codebase. You can implement a code review process where team members take turns reviewing each other's code and providing feedback. Overall, Agile development is a flexible and iterative approach that emphasizes teamwork, collaboration, and communication to deliver high-quality software in a timely manner.

When conflicts arise with colleagues or group members, I believe that communication is key. I think it's important to address the issue directly with the other person and work to find a resolution that best serves the project's goals. This could involve discussing the issue openly and finding a compromise or solution that satisfies both parties.

If the conflict cannot be resolved between the two individuals, I think it may be necessary to escalate the issue to a tech lead or manager who can provide guidance and help mediate the situation. I think it's important to approach the situation professionally and calmly, and to focus on finding a resolution that best serves the project and team.

review code

- Define Standards: Before starting the review, I make sure that we have clearly defined coding standards that everyone on the team follows. This ensures consistency in the codebase and makes the review process more efficient.
- Identify Areas of Focus: I start the review by identifying the areas of code that I want to focus on. This could include new features, bug fixes, or areas of the codebase that are prone to errors.
- Review the Code: Once I've identified the areas of focus, I review the code in detail, looking for any syntax errors, logic flaws, or other issues that could impact the quality or functionality of the code.
- Provide Feedback: When I find issues, I provide feedback to the developer on how to address them. I make sure to provide specific examples and suggest solutions that will help improve the code.
- Collaborate: Code review is a collaborative process, and I encourage developers to ask questions and discuss any issues that they may have. This helps to foster a culture of continuous improvement and ensures that everyone on the team is aligned on the codebase.
- Follow Up: Finally, I make sure to follow up with the developer after the review to ensure that all issues have been addressed and that the code meets our coding standards.

git

clone
pull
fetch
add
commit
push
squash: squash several commits together

cherrypick, pick 1,2 commits from 10 commits

cherrypick, pick 1,2 commits from 10 commits

AWS

- AWS Secrets Manager is a service that helps you protect access to your applications, services, and IT resources. It enables you to securely store and retrieve your secrets such as database credentials, API keys, and other sensitive information. You can use AWS Secrets Manager to protect secrets used by your EC2 instances, RDS databases, S3 buckets, ECS containers, and other AWS resources.
- EC2: When you launch an EC2 instance, you can configure it to retrieve its secrets from AWS Secrets Manager. This ensures that your EC2 instances have access to the required secrets without exposing them in plain text in your code or configuration files.
- RDS: AWS Secrets Manager can be used to securely store and manage credentials for your RDS databases. You can configure your RDS instances to retrieve their credentials from AWS Secrets Manager at runtime.
- S3: You can use AWS Secrets Manager to store and manage access keys for your S3 buckets. This enables you to manage access to your S3 buckets and ensure that only authorized users have access.
- ECS: When you launch an ECS container, you can configure it to retrieve its secrets from AWS Secrets Manager. This ensures that your containers have access to the required secrets without exposing them in plain text in your code or configuration files.

Docker, Kubernetes

we have devops

basic OO question

question you have

SLAs and their management styles.

Can you give me an idea of how much work our team's OE tasks will require in the upcoming sprint? I'd like to understand how much time and effort we should allocate to these tasks, and how they will impact our overall sprint goals and timelines

On average, how many code reviews or pull requests does our team complete in a year? I'm interested in understanding the pace of our code review process and how it may impact our overall productivity and efficiency

I'm curious about the maximum TPS that our team's service API can handle. Could you tell me what the peak TPS has been for our API so far, and how our service architecture is designed to handle spikes in traffic?

Does our team have its own database, or do we share a database with other teams? I'm interested in understanding how our data is managed and how we can ensure that we have the necessary resources to support our database needs